

# Virtual Observatory Publishing with DaCHS

Markus Demleitner, Margarida Castro Neves, Florian Rothmaier, Joachim Wambsganss

Universität Heidelberg, Zentrum für Astronomie, Astronomisches Rechen-Institut, Mönchhofstraße 12-14, 69120 Heidelberg, Germany

## Abstract

The Data Center Helper Suite DaCHS is an integrated publication package for building VO and Web services, supporting the entire workflow from ingestion to data mapping to service definition. It implements all major data discovery, data access, and registry protocols defined by the VO. DaCHS in this sense works as glue between data produced by the data providers and the standard protocols and formats defined by the VO. This paper discusses central elements of the design of the package and gives two case studies of how VO protocols are implemented using DaCHS' concepts.

**Keywords:** virtual observatory, publication tools

**2000 MSC:** 68U35

## 1. Introduction

To aid in the adoption of Virtual Observatory (VO) standards, it is important to keep the entry barrier to running interoperable services low. In particular for the VO's "S-protocols" (SCS<sup>1</sup>, SIAP<sup>2</sup>, SSAP<sup>3</sup>), an important design goal has been a straightforward mapping to common network programming paradigms. Nevertheless, running fully compliant services seamlessly integrated into the Virtual Observatory as a whole requires a significant effort – for example, a registry record needs to be maintained, and the auxiliary services required by VOSI<sup>4</sup> have to be in place as well.

With advanced protocols like TAP<sup>5</sup> the implementation effort is significantly larger. Therefore, packaged standard software that contains all the building blocks necessary for the operation of services in a mutually compatible form is an important contribution to keeping the operation of VO services within reach of modest organisations.

The German Astrophysical Virtual Observatory (GAVO) has been developing such a package since 2007 under the name DaCHS (which stands for Data Center Helper Suite). Compared to most other packages available for this purpose (for instance, VO-Dance (Molinaro et al., 2012) or Saada; for more

information on packages available for VO publishing, see the IVOA's web page on publishing in the VO<sup>6</sup>), its focus is on a unified handling of the entire publication process from the raw data files to the dissemination of data and metadata, including registry records (schematically shown in Fig. 1). DaCHS also helps in organising ancillary tasks that may be necessary to unify data for publication purposes, like header normalization or astrometric calibration as required for effective publication via VO image access protocols. However, actual data reduction tasks are not considered in scope, which means that DaCHS will not, for example, grow an actual workflow engine, and the inputs are expected to be at least almost science-ready.

In DaCHS' development, two main principles served as guidelines:

- (1) Be declarative when reasonably possible
- (2) There is exactly one place for each piece of metadata

Point (1) means that when making design choices, we have a bias towards declarations ("My problem is X") as opposed to procedural definitions ("if a, do b, otherwise c"). This is motivated by the expectation that declarative specifications will allow easier development of the underlying software without requiring adaptation of the service definitions. For an operator running hundreds of services, such a requirement might otherwise block software upgrades indefinitely. On the other hand, for many data publication tasks a procedural specification is much more natural, and DaCHS does take advantage of that. This is why the guideline is qualified by "when reasonably possible".

Point (2) is motivated by the observation that while quality metadata is paramount to smooth VO operation, metadata maintenance rarely gets the attention necessary. Hence, it must be made as easy and labour-efficient as possible. Also, various

<sup>1</sup>Simple Cone Search, a protocol allowing remote searches in catalogs of celestial objects (Williams et al., 2008).

<sup>2</sup>Simple Image Access Protocol, a protocol allowing discovery of images of the sky (Tody and Plante, 2009).

<sup>3</sup>Simple Spectral Access Protocol, a protocol supporting the discovery of spectra (Tody et al., 2012).

<sup>4</sup>Virtual Observatory Support Interfaces, a suite of simple endpoints allowing clients an inspection of a service's data content, access options, and health (Grid and Web Services Working Group, 2011).

<sup>5</sup>The Table Access Protocol (Dowler et al., 2010) enables the exchange of database queries and results between clients and remote servers. For its operation, it depends on several other standards, in particular the SQL-like query language ADQL (Ortiz et al., 2008) and the Universal Worker Service (UWS, Harrison and Rixon 2010) pattern for asynchronous execution.

<sup>6</sup><http://wiki.ivoa.net/twiki/bin/view/IVOA/PublishingInTheVONew>



Glossing over the details, the operator here gives various properties of the table itself (name, primary key, access control), followed by a specification of the structure of space-time coordinates in the table (in this case, this is just a spherical position with errors; see also the STC discussion in section 3). Subsequently, column metadata – names, types, units, UCDs<sup>9</sup> – is given, one element per column.

Finally, the *mixin* attribute in the above fragment deserves more attention. In DaCHS, a *mixin* essentially is a collection of aspects of data representation, first and foremost specific columns, but possibly also pieces of metadata, indices, and the like. Additional behaviour (such as filling the table of file-like products) can optionally be attached to them. Mixins are defined in (typically built-in) resource descriptors and referenced by their identifiers, which in DaCHS consist of the resource descriptor name, a hash, and an XML id. The double slash in front of the resource descriptor name indicates that a built-in resource descriptor is referenced; these come with the software distribution and contain – potentially operator-customisable – material for re-use in operator RDs, as well as descriptions of system services.

The mixins guarantee certain properties in the table that protocols exposing the contained data require. In the example, the mixin ensures that there is one designated spherical position (identified via special UCDs) per table row, and that positional queries over it are fast thanks to Q3C-based indexing (Koposov and Bartunov, 2006). This is what DaCHS’ support for the IVOA SCS protocol builds on.

Another example for a mixin linked to an IVOA protocol is `//siap#pgs` which endows a table with the columns required in responses for SIAP, plus further columns allowing efficient queries over such an image collection using the pgSphere postgres extension. Similar mixins exist for the spectral access protocol, Obscore<sup>10</sup> conformance, and more.

After the definition of the internal representation, the operator next describes the ingestion process, i.e., whatever is necessary to bring the input data as provided by the producer to the internal representation. In DaCHS, ingestion typically is a two-step process. In the first step (corresponding to the top-most arrow in Fig. 1, a parser produces a sequence of mappings (“associative arrays”) from the input files, where usually both keys and values are simple, flat strings. The descriptions defining how a given input file relates to the sequence of mappings in DaCHS are collectively known as grammars, where different types of inputs (FITS files vs. CSV, say) require different sorts of rules. The resulting mappings are called rawdicts (“raw dictionaries”) in DaCHS terminology.

For example, if the input comes as a formatted ASCII table, the grammar would assign labels to column ranges like this:

```
<columnGrammar topIgnoredLines="9">
```

```
<colDefs>
  hipno:      3-8
  srcSel:     47-49
  ...
```

– which instructs the parser for column grammars to ignore the first nine lines, and then, in each line, use the contents of columns 3 through 8 to get the value for the key `hipno`, analogously construct the value for `srcSel`, and so forth.

For comparison, when an input format already is highly structured, as in the case of FITS headers, the grammar specification can be as simple as

```
<fitsProdGrammar/>
```

This tells DaCHS to fairly directly use `pyfits` (Barrett and Bridgman, 1999) to create one rawdict each from (in this case) the primary header of each input file; no additional instructions are necessary in the simplest cases, as the parsing rules for FITS headers are already well-defined by an external standard.

In addition to various sorts of text-based parsers, DaCHS also has parsers built in for FITS binary tables (one rawdict per row), VOTables, or generic binary records. Operators can furthermore write custom parsers in Python.

To actually feed a database table, a second step in the ingestion is necessary, in which the rawdicts are processed to data structures mapping column names to typed (and possibly digested) values. Within DaCHS, these are called rowdicts.

The transformation of rawdicts to rowdicts (the second arrow in Fig. 1) is performed by data mappers (“rowmakers”). This usually involves more than type conversion and key mappings: unit conversions, combining inputs (e.g., date-time values), arbitrary mapping of values (e.g., standardization of object or filter names), detection of NULLs, or computing new values (e.g., waveband limits from filters) are just some of the tasks regularly necessary to ensure compliance to IVOA standards or rationalise data representation.

To support this wide variety of mapping operations, we allow several levels of mixing in procedural content: plain Python expressions as mapping values, applying pre-defined “procedures”, usually passing parameters, or writing such procedures from scratch. Writing complete procedures from scratch obviously is the most expressive method, but as such procedures usually reference implementation details, it also entails the largest potential for incompatibilities as the core software develops. It is much easier for the core software to keep the interface of pre-defined procedures and the namespace visible to the plain Python expression stable.

For instance, a simple rowmaker could look like this:

```
<rowmaker idmaps="*">
  <map key="src_sel" source="srcSel"/>
  <map key="raj2000">hmsToDeg(
    @alphaHMS, None)</var>
</rowmaker>
```

– this would instruct DaCHS to produce values for `src_sel` in the rowdicts from the values of `srcSel` in the incoming rawdicts, converting types as necessary according to default rules,

<sup>9</sup>Unified Content Descriptors, a controlled vocabulary for expressing terms important for expressing astronomical metadata like “pos.eq.ra” or “meta.ref.url” (Derriere et al., 2004).

<sup>10</sup>Obscore is a data model with a specific table schema for observational products; coupled with TAP, it allows complex queries to be run against standardised descriptions of observational products (Louys et al., 2011).

and to use the built-in `hmsToDeg` function to parse the values of `alphaHMS` in the `rawdicts` to obtain the values for `raj2000` in the `rawdicts`. The `idmaps` attribute finally says that all remaining keys from the `rawdict` that have identically-named counterpart in the database table are to be type-converted using default rules.

Table mixins are typically accompanied by `rowmaker` procedures aiding in the mapping from `rawdicts` to the specific columns provided by the mixins. This is an important mechanism for elementary input validation and the provision of defaults. For instance, image metadata necessary for publication over SIAP, together with some elementary manipulation of header values, could be covered by a `rowmaker` like (`@key` is a shortcut notation to access values from the `rawdict`)

```
<rowmaker>
  <var name="cleanedObject">
    @OBJECT.split("_")[0]</var>
  <apply procDef="//siap#setMeta">
    <bind key="title"
      >@cleanedObject+" "+@DATE_OBS</bind>
    <bind key="instrument"
      >"%s %s"%(@TELESCOP, @OBSERVAT)</bind>
    <bind key="dateObs">@DATE_OBS</bind>
```

The `apply` element is a container for procedural python code used in `rowdict` generation. In this instance a predefined procedure (selected, as usual, by a reference into the built-in `//siap` resource descriptor) is invoked with parameters defined in `bind` elements.

Grammar, `rowmaker`, and a specification of what files to read from (typically a shell pattern with optional blacklisting) together form an ingestion rule, which in DaCHS takes a form like

```
<data id="import">
  <sources>data/*.txt</sources>
  <columnGrammar topIgnoredLines="9">
    ...
  <make table="main">
    <rowmaker>
      ...
```

With this, DaCHS has enough information to create and populate the database table. This is effected by executing a command like

```
gavo imp q
```

Here, `gavo` is the name of the DaCHS executable (which is not called something like “dachs” by default for historical reasons). The `imp` argument selects the `import` subcommand, which looks for all data elements in the resource descriptor referenced by its argument and executes the ingestions described there. Finally, `q` is the name of the file containing the resource descriptor – as the canonical extension for resource descriptor files is “`rd`”, this command line would try evaluating a file `q.rd`.

Ideally, this ingestion would only be done once at service creation time. In practice, bugs in the input data, the grammar,

or the mapping rules, evolution of the data collection itself, or improvements in the data publication routinely necessitate re-ingestions, possibly years after a resource has been published. This is the main reason for keeping the ingestion rules in the resource descriptor. Keeping them close to the service and metadata definitions has also proven useful simply for documenting (executably, if need be) what operations have been performed during ingestion.

Even if ingestion typically is a rare event, for data producing more than a few million rows it still may become a bottleneck, as the described operations typically only process a few thousand rows per second on current hardware. This is partly due to both the parser and the mapper being compiled into Python bytecode rather than native object code, partly due to the overhead of going through SQL serialization and deserialization of multi-INSERT statements.

To nevertheless allow rapid ingestion of datasets in the gigarecord range, DaCHS supports a shortcut mechanism called “direct grammar”. These are external binaries creating material going into the database via binary copy in one step. Thus, they sidestep DaCHS’ mapping mechanisms and combine the roles of the normal grammars and the rowmakers in one piece of code. The usual way of creating such external binaries is to use DaCHS’ `mkboost` subcommand to generate C source code templates. When parsing from FITS binary tables, the generated source code will immediately work if the relation between database table and source table is sufficiently simple. In all other cases, manual translation of grammar and mapping rules to C code is required. The reward of the additional effort and the loss of declarativeness is that the ingestion time typically is significantly shorter than the time the database engine spends on indexing and other post-ingestion operations.

The structured representation resulting from the ingestion is the basis for the operation of services. Services adhere to certain protocols that govern the actual bytestreams of the inputs and outputs of the service. That is true regardless of whether the client is a web browser or specialised software speaking IVOA protocols. Most of the protocol logic is hardcoded in so-called *renderers*, which are objects that convert between whatever is on the wire – parameters and uploads on input, serialised byte streams on output – and internal representations – input tables on input, output tables on output. In DaCHS, they are referenced in short strings (“form” for exposing a service over HTML forms, “`siap.xml`” for handling requests using IVOA’s SIAP protocol, “availability” to return VOSI availability information on the service, and so forth).

The actual functionality, based on the internal representations and performing the computations or queries necessary to fulfill the incoming query, is provided by *cores*; the most common one is the `dbCore`, which generates a database query from incoming parameters. For example, here is a simple service definition giving both a web form and an IVOA cone search service on the table sketched above, allowing an additional constraint on a column named `mv`.

```
<service id="cone" allowed="scs.xml,form">
  <dbCore queriedTable="main">
```

```

<FEED source="//scs#coreDescs"/>
<condDesc buildFrom="mv"/>
</dbCore>
<outputTable verbLevel="20"/>
</service>

```

To actually run the services defined in this way, DaCHS has a built-in server component that, during development, is usually operated in its debug mode by executing

```
gavo serve debug
```

The services can now be used, with the access URLs derived from file system paths to the RDs, XML ids within them and renderer names using a simple scheme. To make the services discoverable, however, a publication step is necessary, either locally to the portal page or globally to the VO Registry. A sensible publication also needs carefully written and comprehensive metadata. Within a resource descriptor, metadata comes in dedicated metadata elements, typically at the head of the resource descriptor, for instance:

```

<resource schema="arihip">
  <meta name="title">
    >ARIHIP astrometric catalogue</meta>
  <meta name="description">
    The catalogue ARIHIP has been...
  </meta>
  <meta name="creator.name">
    >Wielen, R....</meta>
  <meta name="subject">Catalogs</meta>...
  <meta name="coverage">
    <meta name="profile">AllSky ICRS</meta>
    <meta name="waveband">Optical</meta>
  </meta>
  <meta name="_longdoc" format="rst">
    The ARIHIP Catalogue is a suitable...
  </meta>
  <meta name="source">2001VeARI..40....1W</meta>

```

For a full publication to the VO, a VOResource (Plante et al., 2008) record needs to be created. This is an XML fragment conforming to a set of XML schema files and collecting a comprehensive and standard set of service metadata. Once the metadata is specified, DaCHS creates those automatically, where capabilities and interfaces declared depend on the renderer used – for instance, the form renderer yields capabilities with interfaces of type `vr:WebBrowser`, while the `ssap.xml` renderer capabilities of type `ssap:SimpleSpectralAccess`. All this is largely transparent to the operator.

Since publications should not happen accidentally, they are two-step processes, in which first one or more publish elements are added in the service element as shown above:

```

<publish render="scs.xml" sets="ivo_managed"/>
<publish render="form"
  sets="ivo_managed,local"/>

```

– meaning that the form-based browser interface is advertised both on the portal (“local”) and to the VO as a whole (“ivo\_managed”), whereas the cone search interface that requires a specialised client can only be located through the VO Registry.

After that,

```
gavo pub arihip/q
```

actually adds the services and data collections marked for publication with the resource descriptor to the set of resources reported to the Registry (where `arihip` here is an operator-chosen path component) or on the portal page. This extra step is non-trivial in that it tests the completeness and, in part, formal correctness of the metadata that goes into the resource record.

The VOResource records generated by DaCHS need to become part of the VO Registry at this point. This happens when a searchable registry harvests it as discussed in Demleitner et al. (2014). To facilitate this harvesting, DaCHS supports the OAI-PMH (Lagoze, 2002) protocol employed by VO registries and thus lets operators run a publishing registry. Once a DaCHS instance is public and some server-global metadata is specified, it can self-register at the registry of registries (Plante, 2007), after which no further operator intervention is necessary for the dissemination of new or updated registry records.

### 3. The Metadata Model

Metadata is the central concept in DaCHS. However, due to a combination of considerations involving re-use of concepts presumably already known to the operator, appropriateness of representation, and ease of implementation, there are several different sources of metadata within DaCHS.

First, there is the usual column metadata like name, type, UCD, description, and the like. Here, DaCHS largely follows the model of VOTable’s FIELD and PARAM elements, adding some attributes as necessary (e.g., suggested table heading, relative importance, hints for value formatting, tags of pertaining table notes).

Second, there is metadata on the space-time coordinates (STC<sup>11</sup>). This includes information on the frame of sets of coordinates within tables as well as the roles (“declination”, “error in proper motion in declination”) played by the columns. DaCHS employs a slightly enhanced version of STC-S<sup>12</sup> for their specification. The introduction of a special handling for this kind of metadata may be vindicated by pointing out that STC-S is also used to convey space-time coverage in RDs and that a key-value representation would be too tedious for human input.

It should be noted that role assignment in other DaCHS-supported VO data models, in particular the spectral data model

<sup>11</sup>The IVOA has a data model for expressing space-time coordinates, their derivatives, and ancillary metadata; see Rots (2007).

<sup>12</sup>STC-S (Rots et al., 2013), the “S” standing for “string”, is a technique in which space-time coordinates organised according to IVOA’s STC data model are written as flat strings.

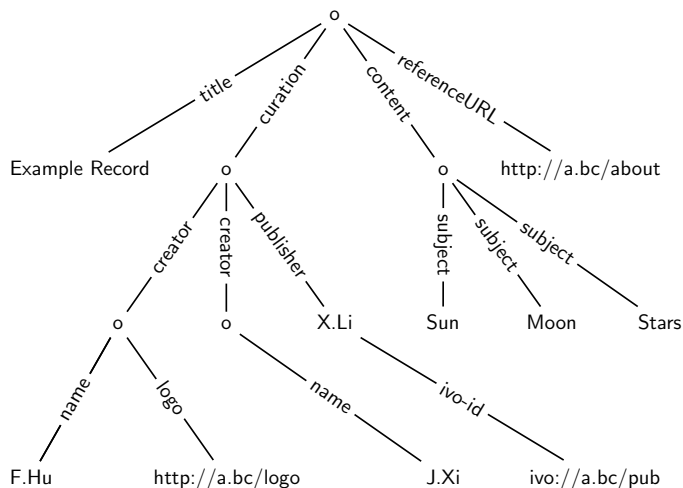


Figure 2: A part of the metadata structure of a resource object within DaCHS.

(SDM; McDowell et al., 2011), so far relies on the `utype`<sup>13</sup> attributes within column metadata. In the context of ongoing attempts to rationalize data model handling in the VO, we expect to develop a unified method to express relations between data models and tables as well as additional model-specific metadata in the future, which should leave STC as less of a special case. We expect, however, to keep special handling for this kind of metadata. This is mainly because of its highly structured nature, the prolific occurrence of references, and the tight coupling to individual pieces of data.

The third and most interesting class of metadata is the “open” metadata. It is used to hold most of the VOResource metadata like title, author, technical contact, related resources or test queries (Hanisch et al., 2007; Plante et al., 2008), as well as locally-used metadata (like detailed documentation, usage hints, or table notes).

Several objects can hold metadata in DaCHS: the whole data center, the resource descriptor, tables, services, and data collections. Between those, metadata can be inherited, in the sense that a piece of metadata requested for a table is first looked up there, then in the table’s resource descriptor, and finally in the data center metadata. This is convenient as it allows setting sensible defaults for items that are typically constant for a data center (the publisher, say) or a resource container (e.g., creator), while still being able to override them in subordinate objects as necessary.

Each meta structure is a labeled tree, where each node contains a single string, can have an unlimited number of outgoing edges, and labels on edges need not be unique. The labels on the edges are called metadata atoms; concatenating the atoms along a path with dots yields conventional metadata labels like “curation.publisher.ivo-id” as used, e.g., in Hanisch et al. (2007).

Figure 2 shows a part of such a metadata tree containing resource metadata as defined by Hanisch et al. (2007). The

reader may take a moment to appreciate the complications of this structure, such as multiple edges with identical labels and inner nodes with values.

This data structure can be fairly straightforwardly mapped to an XML instance document, which is by design: Many pieces of this metadata are used in the creation of the XML-serialised registry records. We refrained from using full DOM trees in the representation for several reasons. From an implementation perspective, our representation is more lightweight; additional logic would be required anyway to provide metadata inheritance; we do not want embedded fragments of VOResource XML with all the intricacies of namespaces in DaCHS’ resource descriptors; and, this data structure is used to hold more than just VOResource metadata.

Table notes, for instance, are metadata on tables. In this scheme, a note consists of the note text in a node with the incoming edge labeled `note`, and there needs to be exactly one outgoing edge labeled `tag`, the node of which contains the note symbol as referenced in the column definitions and used as footnote marks in rendered tables.

This example illustrates two additional requirements that we encountered in applications of DaCHS’ metadata mechanism: Firstly, table notes may require markup, e.g., for definition lists, enumerations, or simply paragraphs. Secondly, certain pieces of metadata depend on others, as in this case a note text is meaningless without the tag.

To satisfy the first requirement, metadata content may be either `literal` (in which no reformatting takes place in usage), `plain` (which reflows text in paragraphs separated by an empty line on input), and `rst`, which interprets the content as ReStructuredText (Goodger, 2010), formatting it according to the format of the embedding document. This also implies that in DaCHS’ metadata component all nodes exclusively contain strings.

The second requirement is covered by a primitive type system for the nodes (rather than their content), which essentially constrains the children of a given node, for instance “note-typed nodes must have a tag child” or “news meta items must have author and date children”. Assigning types to well-known metadata items also enables easy input, type-specific formatting, and defaulting where sensible. Examples where these become relevant include `url` (optionally having a title child), `relation` (having a relationship type and the related resource), `logo` (which, formatted to HTML, yields an `img` element), `bibcode` (which, formatted to HTML, yields links to bibliographic services), and `info` (which are turned into VO-Table INFO elements as appropriate).

A difficult design decision was whether and how to put constraints on meta structures. This is particularly important when meta structures are serialised into VOResource metadata, where adherence to the cardinality rules and content constraints implied by the schema determines the validity of the resource records and hence the validity of the entire OAI-PMH interface.

We opted against enforcing any constraints during metadata construction. The most important reason is that we wanted to enable custom metadata under operator control, as this facilitates straightforward extensibility as well as flexible com-

<sup>13</sup>VOTable’s `utype` attribute was introduced to allow linking entities from data models to elements with VOTable. See Graham et al. (2013) for a discussion of current practices around utypes.

munication between various pieces of operator-provided artefacts (embedded code, HTML templates, and the like). Also incremental creation of meta structures is much easier if unconstrained temporary results are allowed.

On the other hand, it turned out to be highly inconvenient to diagnose and debug bad metadata structures by schema-validating the finished resource records. We therefore introduced a simple specification language that constrains cardinalities of meta items. For the service object, this might look like this:

```
title(1), creationDate(1), description(1),
subject, referenceURL(1), shortName(!)
```

– meaning that exactly one each of title, creationDate, description, referenceURL must be present somewhere in the service’s inheritance tree, one or more of subject must be present, and, as expressed by the exclamation point, exactly one shortName must be in the object’s own metadata. This list is not interpreted as exhaustive, i.e., meta keys not mentioned in this list are simply ignored by the validator. The specifications are checked on demand, typically by executing the `val` subcommand, or before a resource is published to the registry.

No constraints are possible on the node content. While this has not been missed for simple types (e.g., text vs. integer vs. float), violation of constraints imposed by controlled vocabularies has been an issue. As an instructive example, consider SSAP’s creation type, which has to contain one of seven keywords, some of which are long and mixed case, like “spectralExtraction”, as specified in Tody et al. (2012). Weighing the increased complexity in the constraint language against our experience that vocabulary mismatches can quickly be diagnosed by inspecting schema validator outputs, we still decided not to add enumerations of allowed values to the constraint language.

A weak point of the validation-on-demand scheme used by DaCHS is that as the resource descriptors are edited after publication, records may become invalid, since resource records are generated from the descriptions at harvest time, while the OAI-PMH timestamp remains fixed at the time `gavo pub` was executed (which we do to avoid needlessly announcing changed resource records to harvesting registries when registry-irrelevant edits are made to an RD). This can be particularly insidious in the presence of incremental harvesting where the old, correct record will still be present in some registries, while others might have performed a full re-harvest in the meantime and discarded the now malformed record. This situation can only occur if operators fail to re-run `gavo pub` after doing registry-relevant edits – which in practice happens more often than we would like. No good solution for this type of problem exists at that point.

#### 4. Resource Descriptor Techniques

In DaCHS’ philosophy, the information required to publish potentially heterogeneous data over standard protocols is collected in one XML file, the resource descriptor. Typically, a data center will have one resource descriptor per data collection. A single DaCHS server can expose services from arbitrarily many RDs.

As an indicator for the extent of such descriptions, the sizes of the RDs of services currently active in GAVO’s Heidelberg data center vary between 50 lines where essentially only meta-data and a service description is necessary, and 1700 lines for resources with many tables, long notes, and wide integration test coverage<sup>14</sup>.

Rather than discuss the mapping between the model from Fig. 1 and the actual XML elements and attributes in an RD – for that, we refer the reader to Demleitner (2013) and Demleitner (2014) –, we want to present two case studies how the concept of RDs aids the data collection-neutral implementation of VO protocols.

The term “collection-neutral” here is crucial – probably the single most challenging problem in DaCHS’ development and hence the evolution of the definition of RDs has been how to enable optimal use of DaCHS’ facilities while keeping constraints on data published and operator-defined aspects of service behaviour minimal. Given that RDs are written in XML, this translates into mechanisms to re-use and customize subtrees of XML elements. A natural choice for a technology achieving this might seem XSLT, and if we started DaCHS from scratch, XSLT would perhaps play an important role. Historically, though, it took a while until we could state the problem in this form, and so DaCHS now offers custom facilities for this kind of metaprogramming.

One of those is the mixins discussed above. In the current implementation, mixins are largely formulated using a more basic mechanism employing STREAM elements (while all immediate XML names DaCHS defines are mixed-case, metaprogramming element names are written all upper-case), which, at the lowest level, is just a sequence of XML parser events. To allow customization, they support string substitution with macros, inspired by  $\text{\TeX}$  (though much less expressive). For instance, one could define

```
<STREAM id="valWithError">
  <column name="\basename"
    description="\basedesc"
    ucd="\baseucd"/>
  <column name="err_\basename"
    description="Error in
      \decapitalize{\basedesc}"
    ucd="stat.error;\baseucd"/>
</STREAM>
```

This stream can be replayed several times in an RD, where all the macro names not otherwise defined must be “bound” using XML attributes when replaying; this might look like

```
<FEED source="valWithError" basename="mag_v"
  baseucd="phot.mag;em.opt.V">
  <basedesc>The magnitude in the Johnson
  V band, as obtained with the ABC telescope’s
```

<sup>14</sup>Most resource descriptors active in GAVO’s data center are available in a public version control system for inspection and review; a good entry point to this resource is the cross reference at <http://docs.g-vo.org/DaCHS/elemref.html>.

```
1967 V filter.</basedesc>
</FEED>
```

– `basedesc` can be written as an element since DaCHS does not distinguish unique elements with string content and attributes. The `decapitalize` macro used in the stream has not been bound, as built-in macros (in this case lowercasing the first character of its argument) “shine through” the bindings when replaying.

While this basic mechanism may appear rather plain, it supports a surprising breadth of applications, in particular when combined with `LOOP`, which inserts multiple copies of an element into an RD, where macro bindings may be taken from a table or computed using Python code.

An older facility still supported but not recommended for new projects is the `original` attribute – this references an element in some RD, essentially turns it into a stream, and then replays it into the element that contains `original`. While this mimics inheritance known from conventional object-oriented languages fairly well, it turned out that this system had three major drawbacks in RDs. Firstly, there is a mismatch between inheritance and the largely declarative structures in RDs, which frequently made re-use awkward. Secondly, we never found a satisfying syntax for allowing changes of nodes further down the subtree being copied. Thirdly, and most importantly, the source objects for `original` need to be complete, valid DaCHS objects, as they are parsed by the normal RD parser. Useful RD metaprogramming, in contrast, frequently calls for node sets or incomplete objects to be moved around. Streams can do this, `original` cannot.

While streams solved many of the metaprogramming issues we had, more effort is still needed to regularize several important use cases, most of which have to do with element selection; an example are renderer-specific condition descriptors, where, for instance, an HTML form interface will present a free-text field allowing object entry for position selection, where a cone search needs separate RA and DEC inputs. To allow sharing the same core for both renderers, condition descriptors can declare what renderer names they should or should not be used for. Having some general selection in DaCHS’ metaprogramming language could obviate the need for this ad-hoc construct. However, details of such a general selection element, in particular the language the conditions are written in, are non-trivial to define. This results in some rather ugly and ad-hoc DaCHS features at this point.

## 5. Case Study 1: SSAP

The Simple Spectral Access Protocol (Tody et al., 2012) conceptually is comparatively simple: There is one main operation, *queryData*, with a single-table result listing metadata for datasets matching a set of query parameters, expressed using about a dozen protocol-specified and arbitrarily many operator-definable parameters.

However, protocol details add several complications. In addition to responding to the simple data discovery query, the

service also has to declare metadata on the parameters supported by it in a specific format. A wealth of metadata must or should be represented in the response document in either VOTable PARAMs or the response table. Some of the protocol-defined parameters map poorly to many sorts of data collections (e.g., POS to spectra of model atmospheres), and most parameters in turn can support some syntax or have a relatively complex domain. As an example for the latter, we mention the `FORMAT` query parameter, which allows specifying constraints on the format of the datasets returned. While it could simply contain a MIME-type, special values like “compliant” – for result datasets serialised into one of the forms given in McDowell et al. (2011) –, “graphic”, or even “metadata”, as well as combinations of those, must be appropriately processed. Finally, there are some special validity requirements like the declaration of an XML namespace for the fixed prefix `ssa` that is not used in the XML; this latter requirement is due to a now-deprecated convention for data model identifiers.

In consequence, supporting SSAP in DaCHS relies on a fairly complex combination of definitions in system resource descriptors – primarily, the `//ssap` builtin resource descriptor – and a substantial amount of code in the server runtime.

The first step to support SSAP was the definition of the mixins for the tables, i.e., the columns and table parameters making up the metadata collection. We distinguish two cases; in mixin terms these are `//ssap#hcd` and `//ssap#mixc`. The first of these is for publishing “homogeneous” collections of data, in which all datasets originate from the same instrument and indeed the same creator. Consequently, metadata items like the creation type, typical errors, the bibliographic reference or the spectral resolution are table parameters serialised into VOTable PARAM elements. The table itself only has about 20 SSAP-related columns.

The `mixc` mixin is for a “mixed” data collection comprised of spectra from multiple instruments or from multiple observing programs. In this, information kept in table-global parameters within `hcd` tables now moves into each row within the table. This roughly doubles the number of columns (and obviously the effort to fill them from the datasets as well). Both mixins are accompanied by rowmaker procedures to fill the tables in a controlled fashion; their parameter lists double as dataset metadata checklists for the operators.

Other arrangements (e.g., constant instrument data, varying creator data) are of course possible without code changes by simply writing an RD deriving from the built-in `//ssap` RD. However, it is not clear to us yet that the implementation effort for such mixins is worth the moderate savings in space and the moderate gain in data normalization. If a consistent extraction of constant columns into VOTable params were required, we propose this should be done in the VOTable formatting code.

The SSAP service parameters come in the form of a stream, where the code is written in a way that it is irrelevant whether some piece of metadata is in the database table or kept in a table parameter. Implementing those service parameters requires some extra effort, though. About 30 lines of RD are required for support of the `BAND` parameter, as it can contain a floating-point range as well as a string to be compared against bandpass



metadata. More than 50 lines of RD code are necessary to implement standards-compliant behaviour of FORMAT.

For the remaining parameters, generic code was sufficient. The informal specification of what should be supported for the syntax of the parameter values is sometimes called “PQL” or “Parameter Query Language” and basically allows ranges and some sorts of enumerations, plus essentially free additional meta-data which we consistently ignore. It turned out to translate into relatively involved code (about 700 lines of Python and another 120 lines of support objects in an RD).

The built-in //ssap resource descriptor itself has about 800 lines pertaining to SSAP (some material in there deals with generating datasets complying with the IVOA spectral data model).

The actual service interface again requires Python code in the form of a DaCHS renderer. The actual renderer can be shared with the one for the Simple Image Access Protocol SIAP, as error behaviour and the like are defined analogously. The only Python code necessary at that level was the representation of the SimpleDALRegExt (Plante et al., 2012) capability for SSAP services and about 20 lines of relatively declarative code defining how to produce VOResource XML for that capability.

Finally and in contrast to plain SIAP and SCS, SSAP requires a custom core, for example in order to add the gratuitous ssa namespace declaration mentioned above. The total code necessary for the SSAP core is some 20 lines, although it looks much more than that right now since it is still merged with an implementation of a withdrawn proposal for a *getData* operation in SSAP.

## 6. Case Study 2: Datalink

Datalink (Dowler et al., 2013a) is an IVOA protocol currently in development that abstracts dataset delivery by sitting between a discovery query (e.g., using SIAP or TAP queries against Obscore tables) and the actual dataset delivery. In a datalink-enabled scheme, the discovery services may return direct links to results of datalink services, or they may include information on what datalink services can be used to retrieve data from the datasets discovered. Datalink services themselves return a set of links to resources related to the dataset (e.g., the file itself in various formats, previews, raw or further reduced versions) as well as to data access services.

This abstraction is very beneficial to DaCHS; part of the guiding principles behind DaCHS is to touch the inputs as little as possible, which, before datalink, frequently meant brittle solutions to serve standards-compliant (e.g., SDM-compliant spectra) or processed (e.g., cutout) datasets. Datalink now provides a clean framework and lets operators easily define custom operations and complex relationships in their resource descriptors, offloading the core code.

Central to datalink is the concept of a dataset identifier (DID), which is passed into the service to obtain the table of links and services. Most of the DIDs used in the VO are assigned by the publishers, and these are known as pubDIDs for short. PubDIDs should be IVORNs, i.e., a URI with the schema *ivo* assigned to a specific dataset by the operator. As IVORNs must

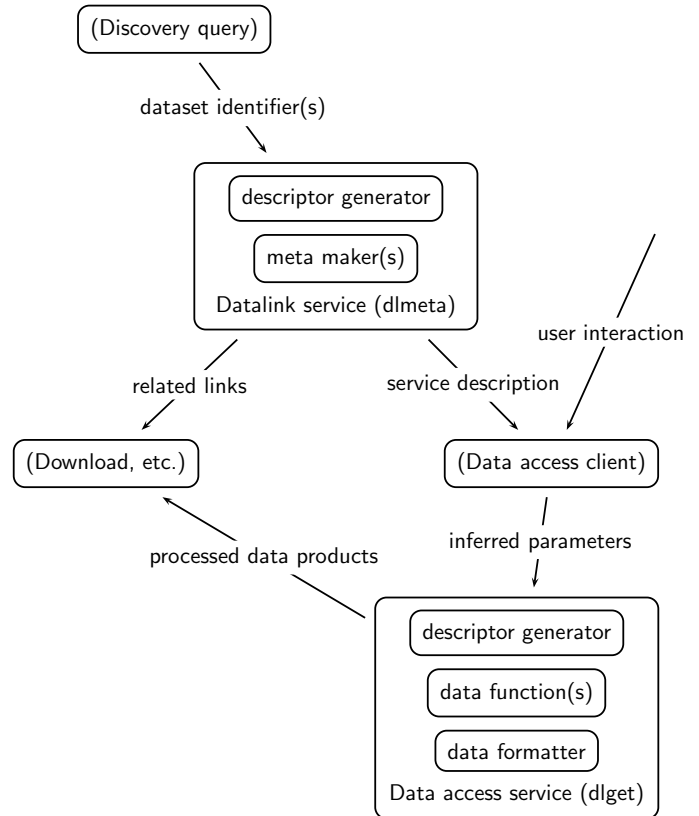


Figure 3: Information flow in datalink-governed data access, with DaCHS actors sketched in. Items in parentheses are external. The datalink service proper (DaCHS *dlmeta* renderer) receives one or more dataset identifiers originating from a previous discovery step. From this, it creates a table of links to actual data products relevant to the dataset, as well as a description of a data access service (in DaCHS, there is at most one of those, exposed through the *dlget* renderer on the datalink service). This allows access to processed (e.g., cutout, rebinned, etc) versions of the dataset.

resolve in the registry (Plante et al., 2007), pubDIDs are typically built as a combination of a registered resource and a query part identifying the dataset itself. The standard pubDIDs of DaCHS consist of the identifier for the product deliverer, having the data center authority as the host part and a tilde as the local part, and a query part giving the access reference, which is a key into the product table. This product table (containing such information as access restrictions, physical paths, or preview location) is part of DaCHS’ subsystem do deal with possibly access-controlled files; by using such access references in the pubDIDs, it is straightforward to re-use as much of DaCHS’ built-in data product handling as appropriate for a given task.

DaCHS’ datalink implementation relies on four new procedure types. The first of these is a *descriptor generator*, which takes a pubDID and generates a descriptor object from it; a default implementation looks up the DaCHS’ product table and returns a bare descriptor containing the content of the matching row, which includes information like the physical location of the data set (e.g., a file path or a URL), the MIME type, and the owner and embargo date for proprietary data.

Other predefined descriptor generators include one looking up SSAP metadata and one reading the primary header of a

FITS file. Both expose this ancillary metadata in extra attributes of the descriptor. Generating descriptors should be fast, as descriptors are generated on each access to the datalink service.

The next procedure type is the *meta maker*. Meta makers receive the descriptor and come up with either parameter definitions for the embedded data access service or link definitions which are later turned into rows of the datalink table. While the meta makers for specialised data products usually presume specialised descriptors, DaCHS does not enforce type safety here.

These two components suffice for building the datalink response. The *dlmeta* renderer formats the objects returned by the meta makers into a VOTable and then delivers it to the client.

In DaCHS, datalink services typically have built-in support for the data access services described. For that, there is the *dlget* renderer, which implements the data access service defined by the meta makers returning service parameters. Processing with this renderer starts as for the *dlmeta* renderer but goes on to pass the descriptor to a sequence of the third type of procedure, the data functions. The first of those plays a special role in that it must add a data attribute to the descriptor containing some representation of the data accessed; this could be a lazy HDU list for FITS files or a table of spectral/flux pairs for a spectrum.

The further *data functions* then perform operations defined by the service parameters on this data, e.g., cutouts, recalibration, or similar. This can happen on the actual data – DaCHS’ spectral processing does this –, but the effect can also be to add processing instructions. This latter option occurs for FITS files, where the data functions handling cutouts just compute slices to be retrieved from disk. This lazy evaluation saves pulling large files like cubes or wide field images into memory just to throw away most of the data.

The final step of *dlget* processing is to call a *data formatter*, the fourth procedure type for datalink support. This takes the content of the descriptor’s data item and serializes it. The default here is trivial: The renderer simply interprets the descriptor’s data attribute as either a file or a pair of MIME type and content (as a byte sequence) and delivers that to the user. In the case of spectral processing, on the other hand, the data formatter serializes the SDM-compliant table in the data attribute into any of VOTable, FITS, CSV, or TSV according to the value of the *FORMAT* attribute.

For special situations, data functions can shortcut the subsequent data functions and the data formatter. This allows the implementation of a *KIND* parameter to the data processing service admitting a value of *header* for FITS files; it will inspect the current state of the data attribute and build a FITS header out of it, which is then immediately rendered.

The implementation effort for datalink itself was rather moderate, in particular not requiring large amounts of handling border cases. The core datalink code is less than 600 lines of Python, about half of which is embedded documentation. Code for handling special data types – namely manipulation of spectra and generic FITS arrays – is 800 lines of RD including documentation.

The whole system proves very flexible. For instance, it

is straightforward to implement delivery for simulated GAIA spectra (Isasi et al., 2010) that came in archives of the GAIA-specific GBIN format. For this, the only thing that needed changing with respect to normal spectral datalink services was about 20 lines of RD describing how to pull the spectral/flux pairs out of the database table generated from the GBIN files.

Another example where datalink enables complex tasks without changing the core code and still remaining compact was a prototype service serving Echelle spectra in about 70 lines of RD.

Using the facilities, a datalink service doing standard cube cutouts is about 20 lines of RD; by way of illustration, this is a slightly abridged excerpt that implements such a cutout for spectral FITS cubes:

```
<service id="d" allowed="dlget,dlmeta">
  <meta name="title">Datalink service...
  <datalinkCore>
    <descriptorGenerator
      procDef="//datalink#fits_genDesc"/>
    <metaMaker
      procDef="//datalink#fits_makeWCSParams"/>
    <dataFunction
      procDef="//datalink#fits_makeHDUList"/>
    <FEED source=
      "//datalink#fits_standardLambdaCutout"
      spectralAxis="1" wavelengthUnit="'nm'"/>
    <dataFunction
      procDef="//datalink#fits_doWCSCutout"/>
    <dataFormatter
      procDef="//datalink#fits_formatHDUs"/>
  </datalinkCore>
</service>
```

A datalink service with links to related datasets in different resolutions and a TAP service containing the cubes in database tables is less than 50 lines of RD.

## 7. Conclusion

DaCHS is a package offering an integrated suite of tools for publishing data with a particular focus on streamlined metadata handling from ingestion to service operation to the generation of registry records. It was written with a view to enabling operators to adapt upstream data to standard interface and implement custom features, always attempting to obtain compliant-by-default behaviour.

The code is freely available under the GPL, ample documentation is provided at <http://docs.g-vo.org/DaCHS>, and we maintain an APT repository, out of which DaCHS can be installed to a state ready to start mapping data within minutes on Debian and derived systems. For further information and downloads, please refer to <http://soft.g-vo.org>.

## Acknowledgements

This work was supported by BMBF grant 05A11VH3.

## References

## References

- Barrett, P. E., Bridgman, W. T., 1999. PyFITS, a FITS Module for Python. In: Mehringer, D. M., Plante, R. L., Roberts, D. A. (Eds.), *Astronomical Data Analysis Software and Systems VIII*. Vol. 172 of Astronomical Society of the Pacific Conference Series. p. 483.
- Demleitner, M., 2013. The DaCHS multi-protocol VO server, *Astronomical Society of the Pacific Conference Series*, in print.
- Demleitner, M., 2014. DaCHS tutorial.  
URL <http://docs.g-vo.org/DaCHS/tutorial.html>
- Demleitner, M., Greene, G., Sidaner, P. L., Plante, R., 2014. The virtual observatory registry, this issue.
- Derriere, S., Gray, N., Mann, R., Preite Martinez, A., McDowell, J., McGlynn, T., Ochsenbein, F., Osuna, P., Rixon, G., Williams, R., 2004. UCD (Unified Content Descriptor) – moving to UCD1+. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/latest/UCD.html>
- Dowler, P., Bonnarel, F., Michel, L., Donaldson, T., Languignon, D., 2013a. Datalink. IVOA Working Draft.  
URL <http://www.ivoa.net/documents/DataLink/>
- Dowler, P., Demleitner, M., Taylor, M., Tody, D., 2013b. Data access layer interface, version 1.0. IVOA Recommendation.  
URL <http://www.ivoa.net/documents/DALI>
- Dowler, P., Rixon, G., Tody, D., Mar. 2010. Table access protocol version 1.0. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/TAP>
- Goodger, D., 2010. reStructuredText, markup syntax and parser component of Docutils.  
URL <http://docutils.sourceforge.net/rst.html>
- Graham, M., Demleitner, M., Dowler, P., Fernique, P., Laurino, O., Lemson, G., Louys, M., Salgado, J., Feb. 2013. UTypes: current usages and practices in the IVOA. IVOA Note.  
URL <http://www.ivoa.net/documents/Notes/UTypesUsage>
- Grid and Web Services Working Group, 2011. IVOA support interfaces version 1.0.  
URL <http://www.ivoa.net/Documents/VOSI/index.html>
- Hanisch, R., IVOA Resource Registry Working Group, NVO Metadata Working Group, 2007. Resource metadata for the Virtual Observatory. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/latest/RM.html>
- Harrison, P., Rixon, G., Oct. 2010. Universal worker service pattern, version 1.0. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/UWS>
- Isasi, Y., Martinez, O., Sartoretti, P., Luri, X., Babusiaux, C., Zaldua, I., September 2010. GOG v8.0 user guide, gAIA-C2-UG-UB-YI-003-8.
- Koposov, S., Bartunov, O., Jul. 2006. Q3C, Quad Tree Cube – The new Sky-indexing Concept for Huge Astronomical Catalogues and its Realization for Main Astronomical Queries (Cone Search and Xmatch) in Open Source Database PostgreSQL. In: Gabriel, C., Arviset, C., Ponz, D., Enrique, S. (Eds.), *Astronomical Data Analysis Software and Systems XV*. Vol. 351 of Astronomical Society of the Pacific Conference Series. p. 735.
- Lagoze, 2002. The open archives initiative protocol for metadata harvesting, version 2.0.  
URL <http://www.openarchives.org/OAI/openarchivesprotocol.html>
- Louys, M., Bonnarel, F., Schade, D., Dowler, P., Micol, A., Durand, D., Tody, D., Michel, L., Salgado, J., Chilingarian, I., Rino, B., de Dios Santander, J., Skoda, P., 2011. Observation data model core components and its implementation in the Table Access Protocol, version 1.0. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/ObsCore/>
- McDowell, J., Salgado, J., Blanco, C. R., Osuna, P., Tody, D., Solano, E., Mazarella, J., D'Abrusco, R., Louys, M., Budavari, T., Dolensky, M., Kamp, I., McCusker, K., Protopapas, P., Rots, A., Thompson, R., Valdes, F., Skoda, P., Rino, B., Cant, J., Laurino, O., 2011. IVOA spectrum data model, version 1.1. IVOA Recommendation.  
URL <http://www.ivoa.net/documents/SpectrumDM/>
- Molinaro, M., Knäpik, C., Smareglia, R., Sep. 2012. The VO-Dance web application at the IA2 data center. In: *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*. Vol. 8451 of Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series.
- Ortiz, I., Lusted, J., Dowler, P., Szalay, A., Shirasaki, Y., Nieto-Santisteba, M. A., Ohishi, M., O'Mullane, W., Osuna, P., the VOQL-TEG, the VOQL Working Group, 2008. IVOA astronomical data query language. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/latest/ADQL.html>
- Plante, R., Jun. 2007. The registry of registries. IVOA Note.  
URL <http://www.ivoa.net/Documents/latest/RegistryOfRegistries.html>
- Plante, R., Benson, K., Graham, M., Greene, G., Harrison, P., Lemson, G., Linde, T., Rixon, G., Stébé, A., Feb. 2008. VOResource: an XML encoding schema for resource metadata version 1.03. IVOA Recommendation.  
URL <http://www.ivoa.net/documents/latest/VOResource.html>
- Plante, R., Delago, J., Harrison, P., Tody, D., May 2012. SimpleDALRegExt: Describing simple data access services, version 1.0. IVOA Proposed Recommendation.  
URL <http://www.ivoa.net/Documents/SimpleDALRegExt>
- Plante, R., Linde, T., Williams, R., Noddle, K., Mar. 2007. IVOA identifiers, version 1.03. IVOA Recommendation.  
URL <http://www.ivoa.net/documents/latest/IDs.html>
- Rots, A., Oct. 2007. STC-S: Space-time coordinate metadata for the Virtual Observatory. IVOA Recommendation.  
URL <http://www.ivoa.net/documents/latest/STC.html>
- Rots, A., Demleitner, M., Dowler, P., 2013. STC-S: Space-time coordinate metadata linear string implementation. IVOA Working Draft.  
URL <http://www.ivoa.net/documents/STC-S/index.html>
- Tody, D., Dolensky, M., McDowell, J., Bonnarel, F., Budavari, T., Busko, I., Micol, A., Osuna, P., Salgado, J., Skoda, P., Thompson, R., Valdes, F., 2012. Simple spectral access protocol version 1.1. IVOA Recommendation.  
URL <http://www.ivoa.net/documents/SSA>
- Tody, D., Plante, R., 2009. Simple image access specification. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/latest/SIA.html>
- Williams, R., Hanisch, R., Szalay, A., Plante, R., 2008. Simple cone search. IVOA Recommendation.  
URL <http://www.ivoa.net/Documents/latest/ConeSearch.html>